Natural Language Processing

Lab 12: Transformer Architecture

Objective

The objective of the lab is to explore, understand and implement the transformer architecture for natural language generation.

Theory

Like the LSTMs, transformers can handle distant information. But unlike LSTMs, transformers are not based on recurrent connections (which can be hard to parallelize), which means that transformers can be more efficient to implement at scale.

Transformers map sequences of input vectors **$(x_1, ..., x_n)$** to sequences of output vectors **$(y_1, ..., y_n)$** of the same length. Transformers are made up of stacks of transformer blocks, each of which is a multilayer network made by combining simple linear layers, feedforward networks, and self-attention layers, the key innovation of transformers. Self-attention allows a network to directly extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in RNNs.

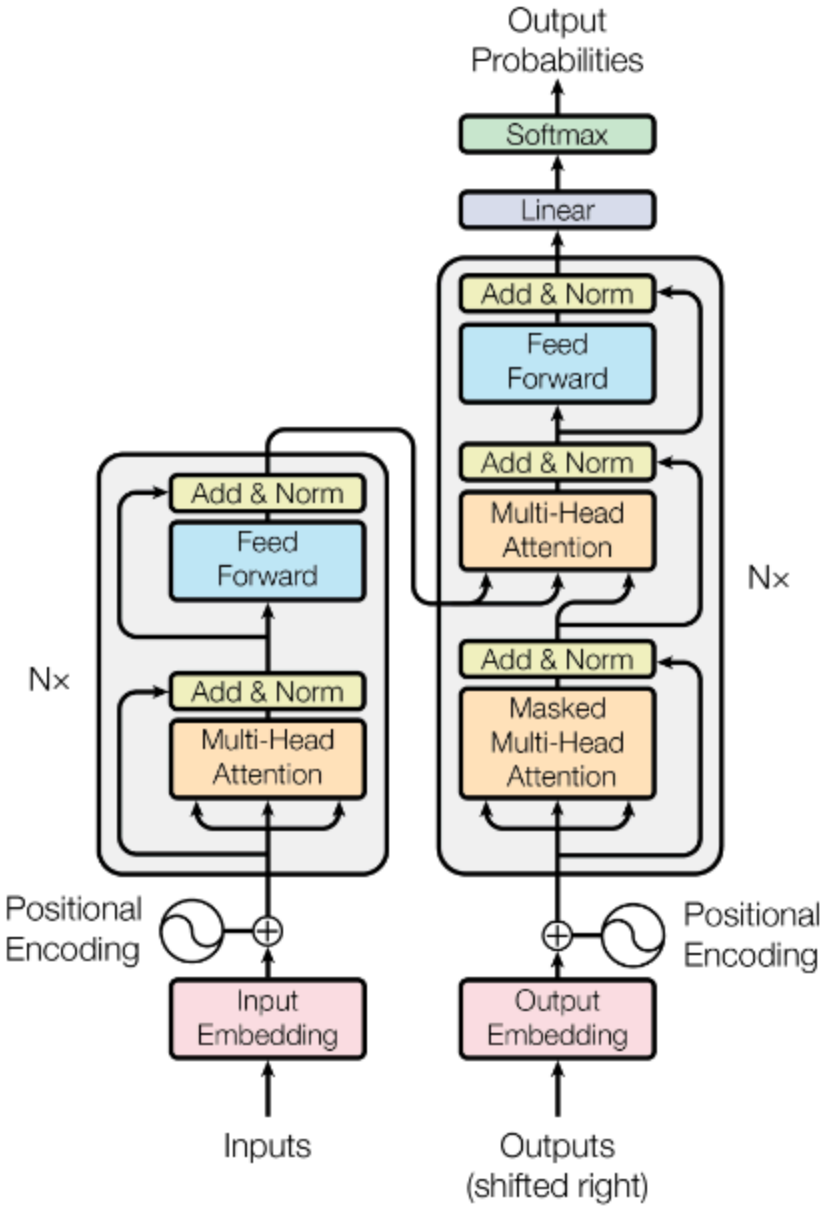


Figure 1: The Transformer - model architecture.

The transformer model consists of 2 parts.

- Encoder
- Decoder

Encoder

An encoder layer is a stack of:

- **Multi-Head Self-Attention Layer:** Contextualizes each word with all others.

- **Feed-Forward Neural Network:** A small, independent network applied to each position (allowing for non-linear transformation).
- **Residual Connections & Layer Normalization:** These are critical for stable training in deep networks. A residual connection adds the original input to the output of a sub-layer ("skip connection"), helping gradients flow. LayerNorm stabilizes the activations.

Input Embeddings

The first step to Transformer inference is to create word embeddings.

Positional Encoding

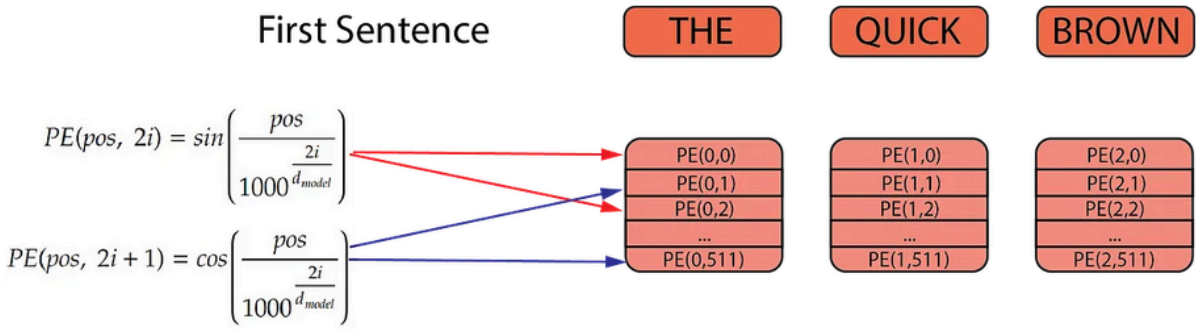
Since the Transformer's self-attention has no inherent notion of word order, we need to inject information about the position of each token in the sequence. Positional Encodings are vectors added directly to the input word embeddings to provide this.

We use the following equations for calculating the positional encoding vector of each word.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where,
pos = position of word whose positional encoding is being calculated.
$\boldsymbol{d_{model}}$ = size of input embedding vector.
i = an iterator which iterates from 0 to $\boldsymbol{d_{model} - 1}$.



The calculated positional encoding vector is then added with the input embedding vector.

Self-Attention Mechanism

Imagine you're reading this sentence:

"The **animal** didn't cross the street because **it** was too tired."

When you see the word "it", your brain instantly connects it to "animal" (not "street"). Self-attention does this mathematically for every word simultaneously.

Instead of processing words sequentially (left-to-right like RNNs), self-attention lets each word look at all words at once to determine which ones matter most for understanding its meaning.

For each word, self-attention creates three vectors by asking:

Query (Q): "What am I looking for?"
(For the word "it": "I need the subject I'm referring to")

Key (K): "What information do I have to offer for matching?"
(For "animal": "I'm a noun, subject of the sentence")

Value (V): "What actual information do I contain?"
(For "animal": "I represent a living organism")

Step 1: Create Q, K, V for each word

```
For each word embedding x_i:
    q_i = x_i * W_Q   # Query vector
    k_i = x_i * W_K   # Key vector
    v_i = x_i * W_V   # Value vector
```

We now have:

```
Q = [q_animal, q_cross, q_it]
K = [k_animal, k_cross, k_it]
V = [v_animal, v_cross, v_it]
```

Step 2: Compute Attention Scores (Who matters?)

For the word "it", we calculate how relevant every other word is by comparing its query with their keys:

```
score("it" → "animal") = q_it · k_animal   (dot product)
score("it" → "cross")  = q_it · k_cross
score("it" → "it")     = q_it · k_it
```

The dot product measures similarity: high score = high relevance.

In matrix form for all words:

```
Scores = Q × K^T  (matrix multiplication)
```

Result is a score matrix where **Scores[i,j]** = how much word **i** attends to word **j**.

Step 3: Scale and Normalize (Convert to probabilities)

Scores are scaled by √**d_k** (where **d_k** is the dimension of **K** vectors) for numerical stability, then passed through softmax:

```
scaled_scores = Scores / √d_k
attention_weights = softmax(scaled_scores, dim=-1)
```

Softmax makes the weights sum to 1 for each word. For "it", we might get:

```
weight("it" → "animal") = 0.8
weight("it" → "cross") = 0.1
weight("it" → "it") = 0.1
```

This means "it" should pay 80% attention to "animal".

Step 4: Weighted Sum of Values (Create new representation)

Now we create the output for "it" by taking a weighted average of all value vectors:

```
output_it = 0.8 * v_animal + 0.1 * v_cross + 0.1 * v_it
```

This output becomes the new, context-aware representation of "it"!

In matrix form for all words:

```
Output = attention_weights × V
```

Multi-Head Attention

Multi-head attention is the mechanism that allows Transformers to simultaneously attend to different types of information from the same input.

Think about understanding this sentence:

"She bought the bank with her savings because she trusted it."

A single attention mechanism might struggle to capture:

- Syntactic relationships (subject-verb: "She bought")
- Semantic relationships (word sense: "bank" as financial institution)
- Coreference resolution ("it" refers to "bank")
- Semantic role ("with her savings" is instrument)

Multi-head attention solves this by having multiple "attention heads" that learn to focus on different patterns in parallel.

Each head will work in a lower-dimensional subspace. For example, if

- d_model = 512 (embedding dimension)
- h = 8 (number of attention heads)
- d_k = d_v = d_model / h = 512 / 8 = 64 (dimension per head)

Instead of one set of Q, K, V projections, we create h separate sets:

For head i (where i = 1 to 8):

```
Q_i = X * W_Q_i   (shape: [seq_len, 64])
K_i = X * W_K_i   (shape: [seq_len, 64])
V_i = X * W_V_i   (shape: [seq_len, 64])
```

where,
seq_len = the number of tokens in the input string

Input X: [seq_len, 512]
    │
    ├── Head 1: Q_1 = X·W_Q_1 → [seq_len, 64]
    ├── Head 2: Q_2 = X·W_Q_2 → [seq_len, 64]
    ├── ...
    └── Head 8: Q_8 = X·W_Q_8 → [seq_len, 64]
Same for Keys and Values.

Compute attention in each head independently. Each head performs independent self-attention:

```
head_i = Attention(Q_i, K_i, V_i)
       = softmax( (Q_i × K_i^T) / √64 ) × V_i
```

Each head_i produces output of shape [seq_len, 64].

Concatenate all heads.

```
multi_head = concat(head_1, head_2, ..., head_8)
           = [head_1, head_2, ..., head_8] along feature dimension
```

Shape: [seq_len, 8 × 64] = [seq_len, 512] (back to original d_model)

Apply one more learned projection to mix information from all heads:

```
output = multi_head × W_O   (W_O shape: [512, 512])
```

Add and Norm

The per token vector output from the Multi-Head Attention layer is then added with the per token positional encoding + embedded vector of the input and normalized via the following.

For a token vector x of dimension d_model:

```
mean = average(x₁, x₂, ..., x_d)
variance = average((x₁ - mean)², ..., (x_d - mean)²)
x_norm = (x-mean)/sqrt(variance+ε)  # ε = small constant (e.g., 1e-5)
output = γ * x_norm + β  # γ and β are learned parameters
```

where,
γ (gamma) = the scale parameter. It's a vector that gets multiplied element-wise with the normalized data.
β (beta) = the shift parameter. It's a vector that gets added element-wise after the scaling.

The output is then fed into a feed forward network which has a hidden layer of 4 times the neural units of the input layer. An Add and Norm operation is applied on the output layer of the FFN and the output of the previous Add and Norm layer.

Task

Create the Encoder block of the Transformer Architecture.